DEVANG PATEL INSTITUTE OF ADVANCE TECHNOLOGY AND RESEARCH

CE261 – Data Structures and Algorithms

Subject Coordinator:

Prof. Khushi Patel, Assistant Professor

Subject Faculties:

Prof. Shraddha Vyas, Prof. Khushi Patel

Devang Patel Institute of Advance Technology And Research
Charotar University of Science and Technology

Fundamentals of Algorithms – Analysis of Algorithm

- The analysis of algorithms involves several techniques and approaches, including:
- **Asymptotic Analysis:** It focuses on characterizing the growth rate of an algorithm's running time and space complexity as the input size approaches infinity.
- Asymptotic notation, such as Big O, Omega, and Theta, is used to describe the upper and lower bounds of an algorithm's efficiency.
- **Worst-case, Average-case, and Best-case Analysis:** Algorithms can have different performance characteristics based on the input they receive.
- Analysis considers the worst-case scenario (input that results in maximum time or space usage), average-case scenario (expected performance on random inputs), and best-case scenario (input that results in minimum time or space usage).
- **Experimental Analysis:** In addition to theoretical analysis, experimental analysis involves running algorithms on actual inputs and measuring their performance. It helps validate theoretical findings and provides practical insights into the algorithm's behavior in real-world scenarios.

Fundamentals of Algorithms – Analysis of Algorithm

- **Best Case Analysis:**
- Best case analysis refers to the analysis of an algorithm's performance under the most favorable or optimal conditions.
- It involves determining the lower bound on the time complexity or other performance metrics when the input to the algorithm is in its best possible state.
- In best case analysis, we consider the input that results in the fewest number of operations or the shortest execution time for the algorithm. It represents an ideal scenario where the algorithm performs exceptionally well.
- It's important to note that best case analysis provides an optimistic view of an algorithm's performance and does not necessarily reflect its typical or average behavior on random inputs.
- It is useful for understanding the algorithm's lower bound in the best possible situation but may not represent the average or worst-case performance in practical scenarios.

Fundamentals of Algorithms – Analysis of Algorithm

- **Example-1**: Let's consider a sorting algorithm such as Quick Sort. In the best case scenario, the input array is already sorted o partially sorted.
- In this case, Quick Sort's partitioning step would divide the array into two roughly equal halves at each recursive call, leading to balanced partitions. As a result, the algorithm's performance is optimal, and the time complexity reduces to its best case, which is typically denoted as O(n log n).
- **Exmaple-2**: Consider a linear search algorithm that searches for a specific element in an array.
- The best case scenario occurs when the target element is found at the very beginning of the array, i.e., at index 0. In this case, the algorithm would perform a single comparison and immediately find the element.
- Therefore, the best case time complexity of linear search is O(1), indicating that it requires a constant number of operations regardless of the size of the input array.

Fundamentals of Algorithms – Analysis of Algorithm

- **Worst Case Analysis:**
- Worst case analysis refers to the analysis of an algorithm's performance under the most unfavorable or pessimistic conditions. It involves determining the upper bound on the time complexity or other performance metrics when the input to the algorithm is in its worst possible state.
- In worst case analysis, we typically consider the input that results in the maximum number of operations or the longest execution time for the algorithm. It represents a scenario where the algorithm performs the least efficiently.
- Worst case analysis provides an upper bound on the time complexity or other performance metrics, guaranteeing that the algorithm will not perform worse than this bound for any input size.
- It helps us understand the maximum resources required by an algorithm, which can be crucial in scenarios where the input data is expected to be large or the algorithm needs to operate efficiently under all possible inputs..

Fundamentals of Algorithms – Analysis of Algorithm

- **Example -1** : Let's consider a sorting algorithm such as Bubble Sort. In the worst case scenario, the input array is in reverse sorted order. In each pass of the algorithm, the largest element shifted to its correct position by swapping adjacent elements.
- In the worst case, the largest element needs to traverse the entire array in each pass, resulting in multiple swaps for each element. This leads to a time complexity of O(n^2) for Bubble Sort in the worst case.
- **Example -2** : Consider a binary search algorithm that searches for a specific element in a sorted array. The worst case scenario occurs when the target element is not present in the array, and the algorithm needs to search until the end of the array.
- In this case, the binary search algorithm repeatedly divides the search space in half and checks the middle element. As the search space is halved at each step, the algorithm requires logarithmic time to reach the end of the array. Therefore, the worst case time complexity of binary search is O(log n), where n is the size of the input array.

Fundamentals of Algorithms – Analysis of Algorithm

- **Average Case Analysis:**
- Average case analysis is a method of analyzing the performance of an algorithm by considering the average behavior over a range of possible inputs. It involves taking into account the statistical distribution of inputs and calculating the expected time complexity or other performance metrics based on that distribution.
- Unlike worst case analysis, which considers the most unfavorable input scenario, and best case analysis, which considers the most favorable input scenario, average case analysis provides a more realistic assessment of an algorithm's performance on typical inputs.

Fundamentals of Algorithms – Analysis of Algorithm

- **Average Case Analysis:**
- Consider a simple linear search algorithm that searches for a specific element in an unsorted array. Let's assume the probability distribution of the input instances follows a uniform distribution, meaning that each element in the array is equally likely to be the target element.
- In this case, the average case analysis involves calculating the expected number of comparisons required to find the target element, taking into account the uniform distribution.
- Let's say we have an array of size n. Since each element has an equal probability of being the target element, the expected number of comparisons can be calculated as the average position of the target element in the array. In a uniform distribution, the average position is (n+1)/2.
- Therefore, the average case time complexity of the linear search algorithm would be O(n/2), which simplifies to O(n).
- This analysis suggests that, on average, the linear search algorithm would need to examine half of the elements in the array before finding the target element when the input follows a uniform distribution.

Fundamentals of Algorithms – Asymptotic Notations

- **Asymptotic notations** are mathematical notations used to describe the upper and lower bounds of the running time or space complexity of algorithms. They provide a way to analyze and compare the efficiency of algorithms without focusing on specific constant factors or lower-order terms. The most commonly used asymptotic notations are:
- **Big O notation (O):** It represents the upper bound or worst-case scenario of the running time or space complexity of an algorithm.
- The notation O(f(n)) denotes that the algorithm's running time or space usage grows no faster than a constant multiple of the function f(n) as the input size n approaches infinity.
- **Omega notation (Ω):** It represents the lower bound or best-case scenario of the running time or space complexity of an algorithm.
- The notation Ω(f(n)) denotes that the algorithm's running time or space usage is at least a constant multiple of the function f(n) as the input size n approaches infinity.

Fundamentals of Algorithms – Asymptotic Notations

- **Theta notation (Θ):** It represents the tight bound or average-case scenario of the running time or space complexity of an algorithm.
- The notation Θ(f(n)) denotes that the algorithm's running time or space usage is bounded both from above and from below by a constant multiple of the function f(n) as the input size n approaches infinity.
- **These notations provide a convenient way to express the growth rates of algorithms and classify them into different complexity classes.**
- For example, if an algorithm has a time complexity of O(n^2), it means that the running time grows quadratically with the input size. If an algorithm has a space complexity of Ω(log n), it means that the space usage grows logarithmically with the input size.
- Asymptotic notations allow us to analyze the scalability and efficiency of algorithms, compare different algorithms, and make informed decisions about which algorithm to use based on their time or space complexity characteristics.

Fundamentals of Algorithms – Sorting Algorithms

- Sorting is any process of arranging items systematically or arranging items in a sequence ordered by some criterion.
- The purpose of sorting is to impose order on a set of data, making it easier to search, retrieve, analyze, and manipulate.
- By organizing the data in a specific order, sorting enables efficient searching and facilitates tasks such as finding the minimum or maximum element, determining if a value exists in the data, or performing various operations that rely on the order of the elements.
- Sorting can be performed on various types of data, including numbers, strings, records, or custom objects.
- The ordering criteria can vary, such as ascending (from smallest to largest) or descending (from largest to smallest) order.
- The choice of sorting algorithm depends on various factors, including the size of the data, the distribution of the data, the stability requirement, and the available resources.

Fundamentals of Algorithms – Bubble Sort

- Bubble sort is a simple comparison-based sorting algorithm.
- It repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order.
- The process is repeated until the entire list is sorted.
- It is one of the simplest sorting algorithms to understand and implement, but it is not very efficient for large or complex data sets.

Fundamentals of Algorithms – Bubble Sort

- **How the bubble sort algorithm works:**
- Start with an unsorted list of elements.
- Compare the first element with the second element.
- If the first element is greater than the second element, swap them.
- Move to the next pair of elements (second and third) and continue comparing and swapping them if necessary.
- Repeat this process until you reach the end of the list.
- At this point, the largest element will be at the end of the list.
- Repeat the above steps for all elements in the list, except for the last one.
- After each iteration, the largest remaining element will "bubble" to the correct position.
- Repeat the process until the entire list is sorted.

Fundamentals of Algorithms – Bubble Sort

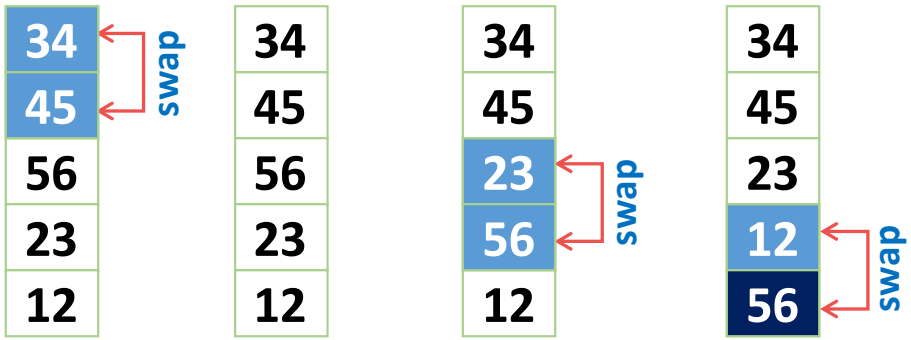
```
# Input: Array A
# Output: Sorted array A

Algorithm: Bubble_Sort(A)
for i ← 1 to n-1 do                    θ(n)
    for j ← 1 to n-i do
        if  A[j] > A[j+1] then
            temp ← A[j]
            A[j] ← A[j+1]
            A[j+1] ← temp              θ(n²)
```

Fundamentals of Algorithms – Bubble Sort

Sort the following array in Ascending order

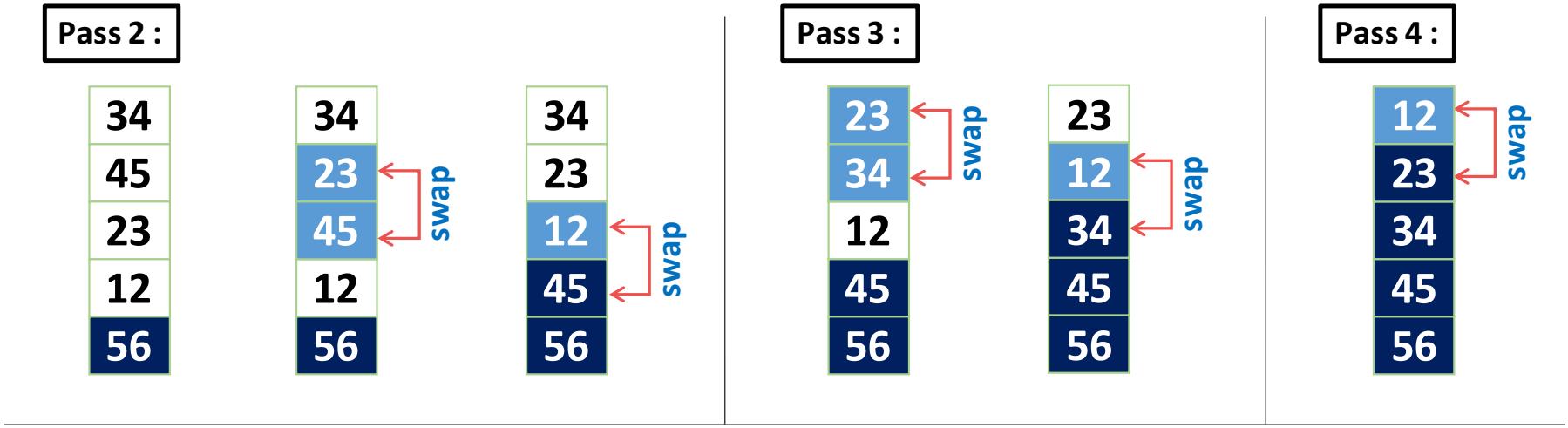
45	34	56	23	12

Pass 1 :



$$if(A[j] > A[j+1])$$
$$swap(A[j], A[j+1])$$

Fundamentals of Algorithms – Bubble Sort



$$if(A[j] > A[j+1])$$
$$swap(A[j], A[j+1])$$

Fundamentals of Algorithms – Selection Sort

- Selection sort is a simple comparison-based sorting algorithm.
- It works by dividing the input array into two parts: the sorted part and the unsorted part.
- The algorithm repeatedly selects the smallest (or largest, depending on the sorting order) element from the unsorted part and moves it to the beginning of the sorted part.
- This process continues until the entire array is sorted.
- **How the selection sort algorithm works?**
- Start with an unsorted array of elements.
- Find the smallest (or largest) element in the unsorted part of the array.
- Swap the smallest element with the first element of the unsorted part, moving it to the sorted part.
- Increment the boundary of the sorted part and decrement the boundary of the unsorted part.
- Repeat steps 2-4 until the entire array is sorted.

Fundamentals of Algorithms – Selection Sort

```
# Input: Array A
# Output: Sorted array A

Algorithm: Selection_Sort(A)
for i ← 1 to n-1 do
      minj ← i;
      minx ← A[i];
      for j ← i + 1 to n do
            if A[j] < minx then
                  minj ← j;
                  minx ← A[j];
      A[minj] ← A[i];
      A[i] ← minx;
```

Fundamentals of Algorithms – Selection Sort

Sort the following elements in Ascending order

5	1	12	-5	16	2	12	14

Step 1 :

Unsorted Array

5	1	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

Step 2 :

Unsorted Array (elements 2 to 8)

-5	1	12	5	16	2	12	14
1	2	3	4	5	6	7	8

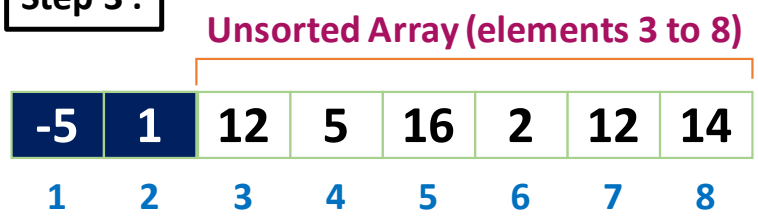
Swap

- **Minj** denotes the current index and **Minx** is the value stored at current index.
- **So, Minj = 1, Minx = 5**
- Assume that currently **Minx** is the smallest value.
- Now find the smallest value from the remaining entire Unsorted array.

Index = 4, value = -5

Fundamentals of Algorithms – Selection Sort

Step 3 :

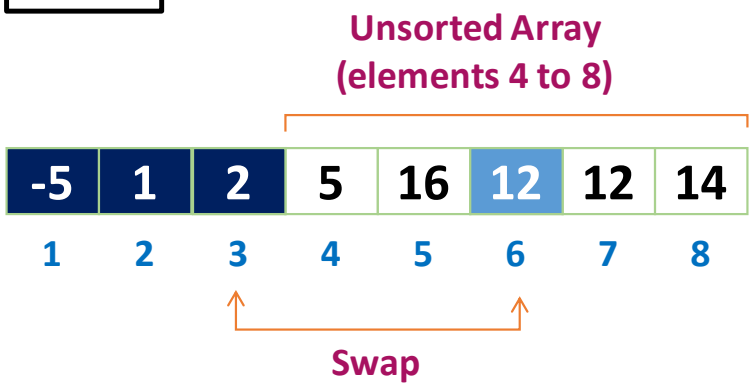


- **Now Minj = 2, Minx = 1**
- Find min value from remaining unsorted array

Index = 2, value = 1

No Swapping as min value is already at right place

Step 4 :

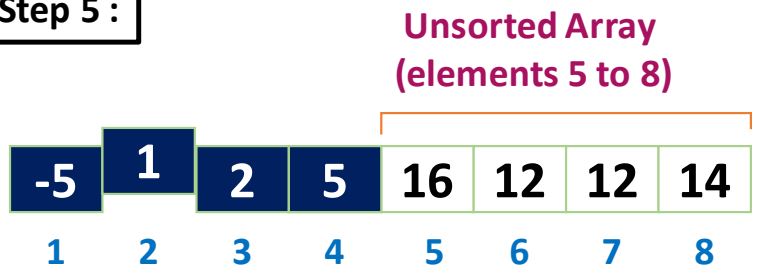


- **Minj = 3, Minx = 12**
- Find min value from remaining unsorted array

Index = 6, value = 2

Fundamentals of Algorithms – Selection Sort

Step 5 :

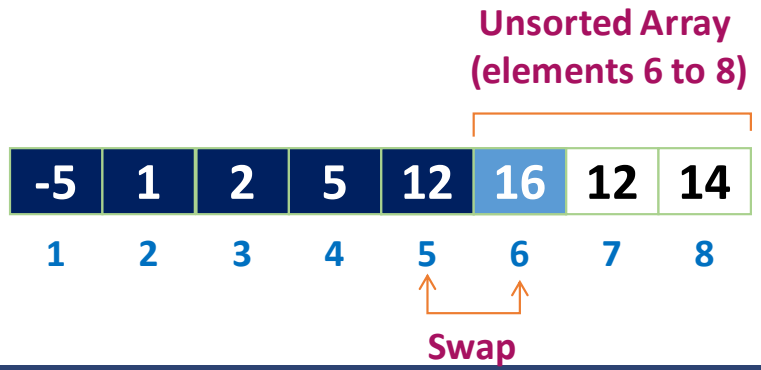


- **Now Minj = 4, Minx = 5**
- Find min value from remaining unsorted array

Index = 4, value = 5

No Swapping as min value is already at right place

Step 6 :

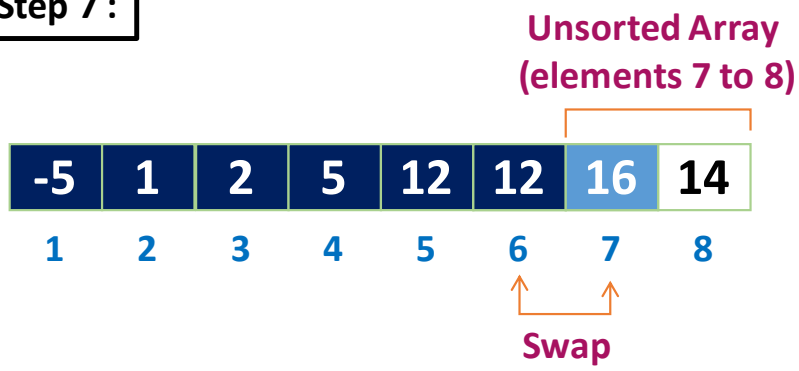


- **Minj = 5, Minx = 16**
- Find min value from remaining unsorted array

Index = 6, value = 12

Fundamentals of Algorithms – Selection Sort

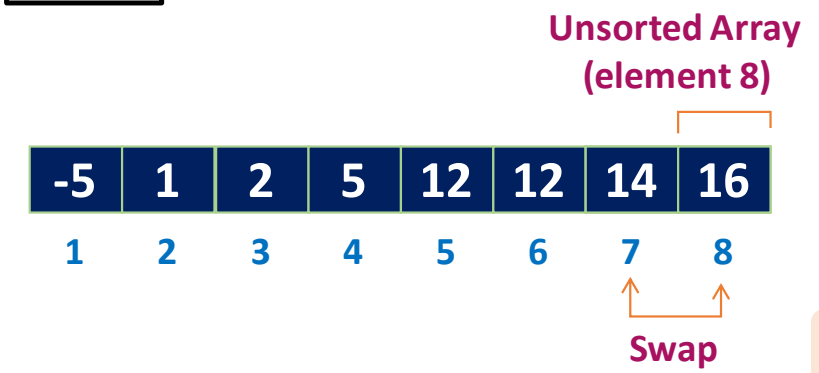
Step 7 :



- **Now Minj = 6, Minx = 16**
- Find min value from remaining unsorted array

Index = 7, value = 12

Step 8 :



- **Minj = 7, Minx = 16**
- Find min value from remaining unsorted array

Index = 8, value = 14

The entire array is sorted now.

Fundamentals of Algorithms – Insertion Sort

- Insertion sort is a simple comparison-based sorting algorithm that builds the final sorted array one element at a time.
- It works by dividing the input array into two parts: the sorted part and the unsorted part.
- The algorithm iterates over the unsorted part, taking each element and inserting it into its correct position within the sorted part.

Fundamentals of Algorithms – Insertion Sort

- How the insertion sort algorithm works:

- Start with an unsorted array of elements.
- Assume the first element is already in the sorted part (considered as the sorted subarray of size 1).
- Iterate over the remaining unsorted elements from the second element (index 1) to the last element (index n-1), where n is the size of the array.
- In each iteration, compare the current element with the elements in the sorted part from right to left.
- Shift the elements in the sorted part that are greater than the current element one position to the right.
- Insert the current element into its correct position within the sorted part.
- Repeat steps 3-6 until the entire array is sorted.

Fundamentals of Algorithms – Insertion Sort

```
# Input: Array T
# Output: Sorted array T

Algorithm: Insertion_Sort(T[1,…,n])
for i ← 2 to n do
        x ← T[i];
        j ← i - 1;
        while x < T[j] and j > 0 do
                T[j+1] ← T[j];
                j ← j - 1;
        T[j+1] ← x;
```

$\theta(n)$

$\theta(n^2)$

Fundamentals of Algorithms – Insertion Sort

Sort the following elements in Ascending order

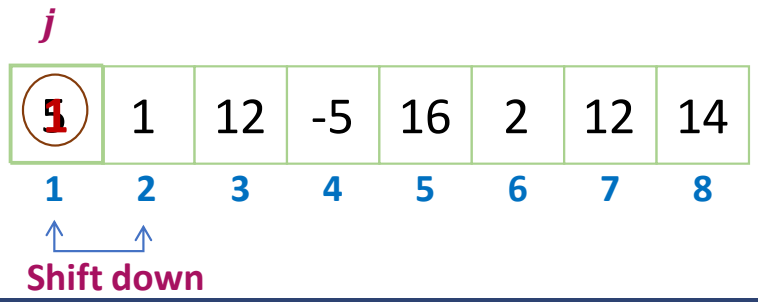
5	1	12	-5	16	2	12	14

Step 1 :

Unsorted Array

5	1	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

Step 2 :

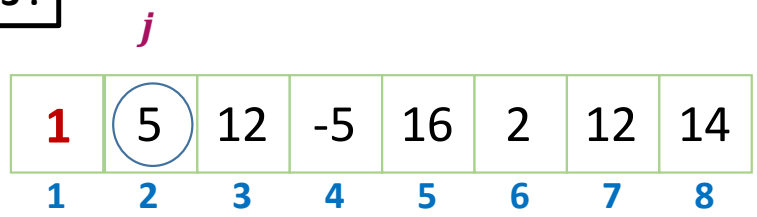


$i = 2, x = 1$ | $j = i - 1$ *and* $j > 0$

```
while x < T[j] do
    T[j + 1] ← T[j]
    j − −
```

Fundamentals of Algorithms – Insertion Sort

Step 3 :

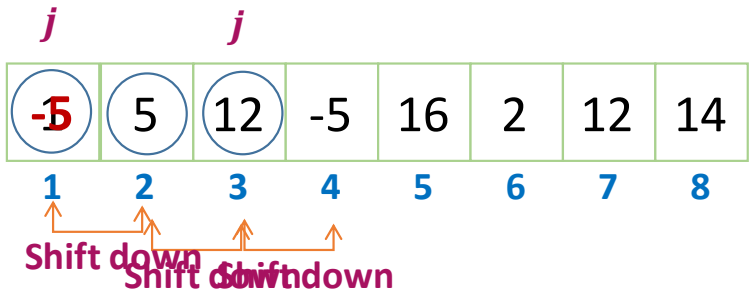


$i = 3, x = 12$ | $j = i - 1$ *and* $j > 0$

```
while x < T[j] do
        T[j + 1] ← T[j]
        j − −
```

No Shift will take place

Step 4 :

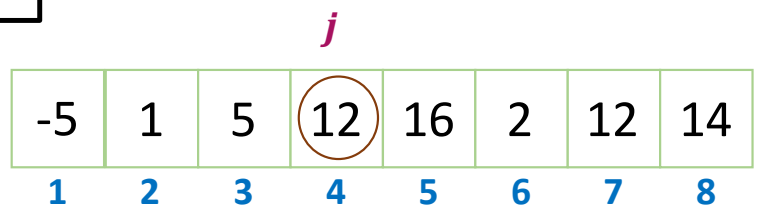


$i = 4, x = -5$ | $j = i - 1$ *and* $j > 0$

```
while x < T[j] do
        T[j + 1] ← T[j]
        j − −
```

Fundamentals of Algorithms – Insertion Sort

Step 5 :

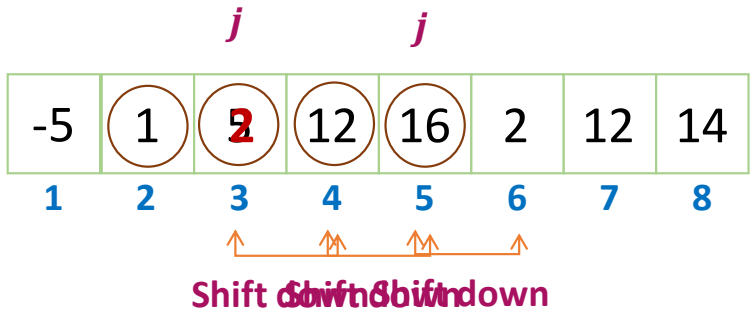


No Shift will take place

$i = 5, x = 16$ | $j = i - 1$ and $j > 0$

```
while x < T[j] do
        T[j + 1] ← T[j]
        j − −
```

Step 6 :

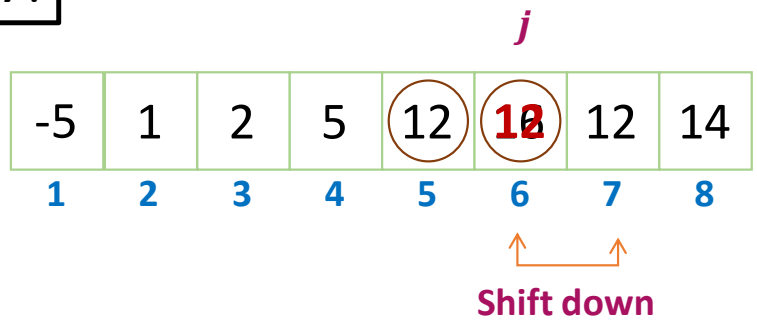


$i = 6, x = 2$ | $j = i - 1$ and $j > 0$

```
while x < T[j] do
        T[j + 1] ← T[j]
        j − −
```

Fundamentals of Algorithms – Insertion Sort

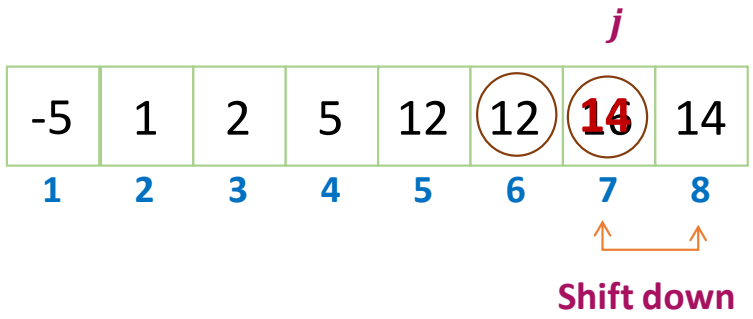
Step 7 :



$i = 7, x = 12$ | $j = i - 1$ and $j > 0$

```
while x < T[j] do
        T[j + 1] ← T[j]
        j − −
```

Step 8 :



$i = 8, x = 14$ | $j = i - 1$ and $j > 0$

```
while x < T[j] do
        T[j + 1] ← T[j]
        j − −
```

The entire array is sorted now.

Thank You